EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Masterthesis Computer Science

PGcuckoo — Inject physical plans into PostgreSQL

Denis Hirn

September 23, 2018

Supervisor

Prof. Dr. Torsten Grust

Co-Supervisor

Prof. Dr. Klaus Ostermann

Hirn, Denis:
PGcuckoo — Inject physical plans into PostgreSQL
Masterthesis Computer Science
Eberhard Karls Universität Tübingen
From July 01, 2018 to December 31, 2018

Abstract

Plan forcing is the most capable plan hint that a database system can implement. It allows the specification of almost every aspect of an execution plan. The open source database system PostgreSQL does not implement any plan hints by default. This thesis shows how an extension can be used to provide plan forcing for PostgreSQL. In addition to that we present an inference rule system that assists and simplifies the process of creating artificial execution plans.

Contents

ACRONYMS

SQL	Structured Query Language
RDBMS	Relational Database Management System
DSL	Domain Specific Language
AST	Abstract Syntax Tree
SPI	Server Programming Interface

1 INTRODUCTION

> Relational query languages provide a high-level *declarative* interface to access data stored in a Relational Database Management System (RDBMS). Over time, SQL has emerged as the standard for relational query languages. Two key components of the query evaluation component of a SQL database system are the *query optimizer* and the *query execution engine*.
>
> The query execution engine implements a set of *physical operators*. An operator takes as input one or more data streams and produces an output data stream. Examples of physical operators are (external) sort, sequential scan, index scan, nested-loop join and sort-merge join. An abstract representation of such an execution is a *physical operator tree*. The execution engine is responsible for the execution of the plan that results in generating answers to the query. Therefore, the capabilities of the query execution engine determine the structure of the operator trees that are feasible.
>
> The query optimizer is responsible for generating the input for the execution engine. It takes a parsed representation of a SQL query as input and is responsible for generating an *efficient* execution plan for the given SQL query from the space of possible execution plans. The task of an optimizer is nontrivial since for a given SQL query, there can be a large number of possible operator trees.
>
> Furthermore, the throughput or the response times for the execution of these plans may be widely different. Therefore, a judicious choice of an execution by the optimizer is of critical importance. [1]

If it is computationally feasible, the query optimizer will examine each of the possible plans, ultimately selecting the execution plan that is expected to run the fastest. [3, 50.5. Planner/Optimizer]

> PostgreSQL's disk-oriented cost model combines CPU and I/O costs with certain weights. Specifically, the cost of an operator is defined as a weighted sum of the number of accessed disk pages (both sequential

> and random) and the amount of data processed in memory. The cost of a query plan is then the sum of the costs of all operators. The default values of the weight parameters used in the sum (cost variables) are set by the optimizer designers and are meant to reflect the relative difference between random access, sequential access and CPU costs. [5, 5.1 The PostgreSQL Cost Model]
>
> Using cardinality estimates as its principal input, the cost model then chooses the cheapest alternative from semantically equivalent plan alternatives. Theoretically, as long as the cardinality estimates and the cost models are accurate, this architecture obtains the optimal query plan. In reality, cardinality estimates are usually computed based on simplifying assumptions like uniformity and independence. In real-world data sets, these assumptions are frequently wrong, which may lead to sub-optimal and sometimes disastrous plans. [5, 1. Introduction]

The database system can be guided to find a good plan by creating indexes on tables or through query rewriting, but we can not *force* a specific execution plan.

1.1 Purpose

The purpose of this thesis is to create the ability to design and compile physical plans *directly* – no SQL or SQL compilation involved. The RDBMS PostgreSQL is well suited for this modification, because it is open source and widely known for its extensibility. This enables the design of PostgreSQL code generators that do not rely on SQL serialization and re-parsing.

The overall structure of this thesis takes the form of six chapters, including this introductory chapter. Chapter 2 begins by presenting the essential data structures of which a physical plan consists. The most important physical operators are defined and explained. The functionality to inject physical plans into PostgreSQL will be established in chapter 3 using an extension. Chapter 4 presents the Haskell back-end, which is the main result of this thesis. The back-end uses an *inference rule system* to simplify the creation of artificial physical plans. Chapter 5 analyses the results of an experiment based on the paper *Unnesting Arbitrary Queries* by Thomas Neumann and Alfons Kemper. [6] Finally, the conclusion in chapter 6 gives a brief summary and critique of the findings.

2 PHYSICAL ALGEBRA OF POSTGRESQL

An essential part of this thesis is the development of a tool, which supports the creation of artificial execution plans, that are eventually compatible with the PostgreSQL *executor*. Therefore, we need to understand how the planner/optimizer arranges the input for the executor.

This chapter gives a brief explanation of the most important physical operators and the corresponding *plan tree* data structures. The `plan tree` serves as input for the executor and is generated by the planner/optimizer. Each node specifies a physical operator and contains any information the executor needs for the evaluation.

The logical basis of a Relational Database Management System (RDBMS) is usually an extended version of the *relational algebra* by Edgar F. Codd. An implementation of relational algebra operations is called *physical algebra*. In general, there is no bijective mapping between a SQL query and physical operators because non-trivial SQL queries can be evaluated by numerous semantically identical plans.

PostgreSQL performs several steps to transform a SQL query from the textual representation into a plan tree. The first step involves to *parse* the query, which generates the *parse tree*. Following this, the *analyzer* performs a semantic analysis of the parse tree and creates a new representation called *query tree*. The *planner/optimizer* uses the `query tree` to generate the most efficient `plan tree`. In the end, the executor evaluates the plan tree. These steps are illustrated in figure 2.1.

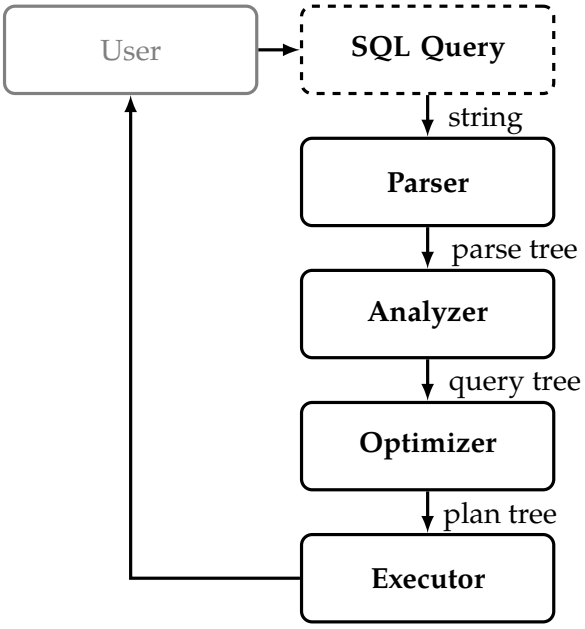

Figure 2.1: Basic transformation steps PostgreSQL performs when a SQL query is received.

2.1 Query Plan Structures

This section shows and explains the definitions of the most important data structures of physical plans. The code snippets of the data type definitions are truncated because a full discussion of all data fields is beyond the scope of this thesis. The complete definitions can be found in the file `plannodes.h` of the PostgreSQL source code. [4]

The plan tree nodes of PostgreSQL are defined using language-`C` structures. A `C` *struct* defines a composite data type that can contain other complex and simple data types. *Inheritance* is achieved by convention. Fig. 2.2 shows how the inheritance hierarchy of the plan tree structures is organized.

2.1.1 PlannedStmt

> The output of the planner is a Plan tree headed by a PlannedStmt node. PlannedStmt holds the "one time" information needed by the executor. [4, plannodes.h, 30–34]

Lst. 2.3 shows the relevant data fields of `PlannedStmt` that are used for *select*-statements. Insert-, update-, delete- as well as utility-statements are not covered by this thesis.

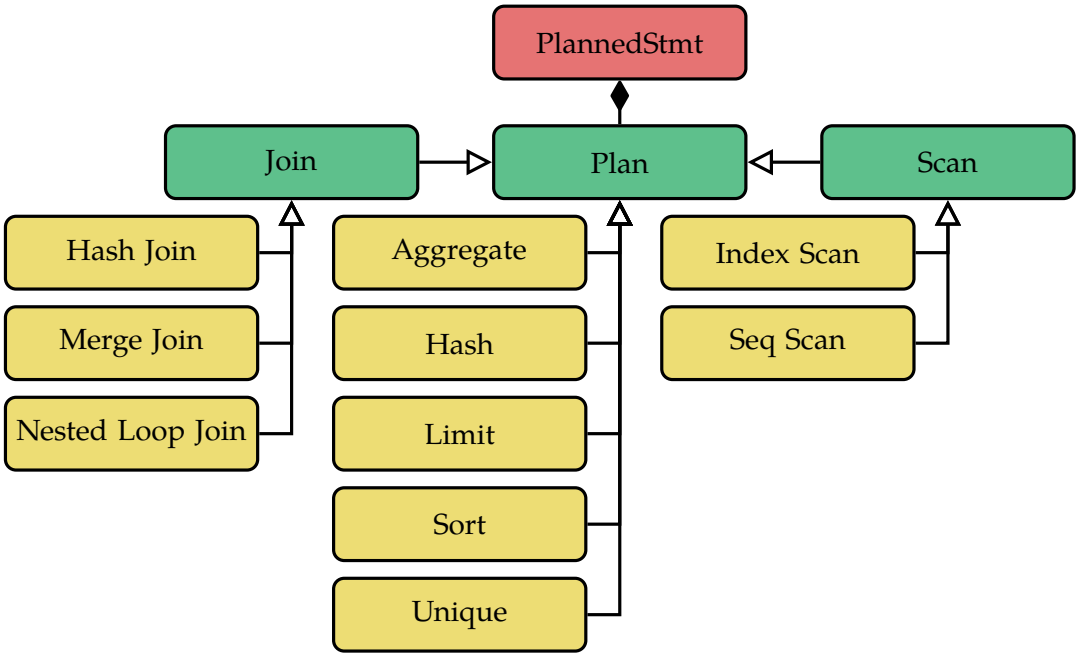

Figure 2.2: PostgreSQL plan tree structure

A Scan-operation is used to access a relation. The information about these relations is stored in the `rtable` list of the `PlannedStmt` node instead of in the Scan-operator.

> The range table is a list of relations that are used in a query. In a `SELECT` statement these are the relations given after the `FROM` keyword. [3, 40.1 The Query Tree]

```
typedef struct PlannedStmt
{
  CmdType    commandType; /* select|insert|update|delete|utility */
  . . .
  bool   parallelModeNeeded;  /* parallel mode required to execute? */
  struct Plan *planTree;      /* tree of Plan nodes */
  List        *rtable;        /* list of RangeTblEntry nodes */
  . . .
  List   *subplans;    /* Plan trees for SubPlan expressions; note
                        * that some could be NULL */
  . . .
  int    nParamExec;   /* number of PARAM_EXEC Params used */
  . . .
} PlannedStmt;
```

Listing 2.3: Definition of PlannedStmt [4, plannodes.h, 41–99]

2.1.2 Plan

> All plan nodes "derive" from the *Plan* structure by having the Plan structure as the first field. This ensures that everything works when nodes are cast to Plan's. (node pointers are frequently cast to Plan* when passed around generically in the executor) We never actually instantiate any Plan nodes; this is just the common abstract superclass for all Plan-type nodes. [4, plannodes.h, 106–117]

The *Plan* structure is the *superclass* of all physical operators and defines the basic data fields all operators have in common. Generally, each operator can have zero, one or two other plan trees as input. These inputs are stored in the `lefttree` and `righttree` field of a `Plan`. The convention is that the "left" plan is the "outer" plan and the "right" plan is the "inner" plan. Most operators can apply a filter predicate (such as the **WHERE**-clause of a SQL query) to the output data stream. These conditions are represented as a list of implicitly-ANDed expressions called `qual`. The `targetlist` stores the target list items. Each target list item defines a *column* of the output data stream.

An exception to the left- and righttree pattern is the *Append* operator. `Append` takes the results of n plans and generates the concatenation. The executor ignores input from the left- or righttree field in this case.

```
typedef struct Plan
{
  . . .
  /* information needed for parallel query */
  bool         parallel_aware; /* engage parallel-aware logic? */
  bool         parallel_safe;  /* OK to use as part of parallel plan? */
  /* Common structural data for all Plan types. */
  int          plan_node_id; /* unique across entire final plan tree */
  List        *targetlist; /* target list to be computed at this node */
  List        *qual;       /* implicitly-ANDed qual conditions */
  struct Plan *lefttree;   /* input plan tree(s) */
  struct Plan *righttree;
  List        *initPlan;   /* Init Plan nodes (un-correlated expr
                            * subselects) */
  /* Information for management of parameter-change-driven rescanning */
  Bitmapset  *extParam;
  Bitmapset  *allParam;
} Plan;
```

Listing 2.4: Definition of the Plan structure [4, plannodes.h, 118–164]

2.1.3 Scan

A *Scan*-operation is used to access *range table entries*. A range table entry may represent a plain relation, a sub-select or a *common table expression*, for example. The planner/optimizer generates the `rtable`-list and assigns the *index* (the position in the list) of the corresponding range table entry to the `Scan` operator.

```
typedef struct Scan
{
    Plan    plan;
    Index   scanrelid;  /* relid is index into the range table */
} Scan;
```

Listing 2.5: Scan structure [4, plannodes.h, 326–330]

2.1.4 Seq Scan

> The *Seq Scan* operator is the most basic query operator. Any single-table query can be carried out using the `Seq Scan` operator.
>
> `Seq Scan` works by starting at the beginning of the table and scanning to the end of the table. For each row in the table, `Seq Scan` evaluates the query constraints (that is, the `WHERE` clause); if the constraints are satisfied, the required columns are added to the result set. [...]
>
> The planner/optimizer chooses a `Seq Scan` if there are no indexes that can be used to satisfy the query. A `Seq Scan` is also used when the planner/optimizer decides that it would be less expensive (or just as expensive) to scan the entire table and then sort the result set to meet an ordering constraint (such as an `ORDER BY` clause). [2, p. 151]

The `Seq Scan` operator is defined as: `typedef Scan SeqScan`. This operator can be executed in parallel. The plan fields `parallel_aware` and `parallel_safe` as well as the flag `parallelModeNeeded` of the `PlannedStmt` must be set to `True` for parallel execution.

2.1.5 Index Scan

> An *Index Scan* operator works by traversing an index structure. If you specify a starting value for an indexed column (`WHERE record_id >= 1000`, for example), the `Index Scan` will begin at the appropriate value. If you specify an ending value (such as

`WHERE record_id < 2000`), the `Index Scan` will complete as soon as it finds an index entry greater than the ending value.

The planner/optimizer uses an `Index Scan` operator when it can reduce the size of the result set by traversing a range of indexed values, or when it can avoid a sort because of the implicit ordering offered by an index. [2, pp. 151–152]

```
typedef struct IndexScan
{
    Scan  scan;
    Oid   indexid;   /* OID of index to scan */
    List *indexqual; /* list of index quals (usually OpExprs) */
    . . .
} IndexScan;
```

Listing 2.6: IndexScan structure definition [4, plannodes.h, 386–396]

2.1.6 Sort

The *Sort* operator imposes an ordering on the result set. PostgreSQL uses two different sort strategies: an in-memory sort and an on-disk sort. You can tune a PostgreSQL instance by adjusting the value of the `sort_mem` run time parameter. If the size of the result set exceeds `sort_mem`, `Sort` will distribute the input set to a collection of sorted work files and then merge the work files back together again. If the result set will fit in `sort_mem` × 1024 bytes, the sort is done in memory using the QSort algorithm. [2, p. 152]

```
typedef struct Sort
{
    Plan        plan;
    int         numCols;       /* number of sort-key columns */
    AttrNumber *sortColIdx;    /* their indexes in the target list */
    Oid        *sortOperators; /* OIDs of operators to sort them by */
    Oid        *collations;    /* OIDs of collations */
    bool       *nullsFirst;    /* NULLS FIRST/LAST directions */
} Sort;
```

Listing 2.7: Sort structure definition [4, plannodes.h, 742–750]

2.1.7 Unique

> The *Unique* operator eliminates duplicate values from the input set. The input set must be ordered by the columns, and the columns must be unique. [...]
>
> The `Unique` operator removes only rows–it does not remove columns and it does not change the ordering of the result set. [2, p. 152]

```
typedef struct Unique
{
    Plan        plan;
    int         numCols;       /* number of columns to check */
    AttrNumber *uniqColIdx;    /* their indexes in the target list */
    Oid        *uniqOperators; /* equality operators to compare with */
} Unique;
```

Listing 2.8: Unique structure definition [4, plannodes.h, 818–824]

2.1.8 Limit

> The *Limit* operator is used to limit the size of a result set. PostgreSQL uses the `Limit` operator for both `Limit` and `Offset` processing. The `Limit` operator works by discarding the first x rows from its input set, returning the next y rows, and discard the remainder. [2, p. 153]

```
typedef struct Limit
{
  Plan  plan;
  Node *limitOffset; /* OFFSET parameter, or NULL if none */
  Node *limitCount;  /* COUNT parameter, or NULL if none */
} Limit;
```

Listing 2.9: Limit structure definition [4, plannodes.h, 921–926]

2.1.9 Aggregate

> The planner/optimizer produces an *Aggregate* operator whenever the query includes an aggregate function. [...]
>
> `Aggregate` works by reading all the rows in the input set and computing the aggregate values. If the input set is not grouped, `Aggregate` produces a single result row. [2, p. 153]

An Agg node implements plain or grouped aggregation. For grouped aggregation, we can work with presorted input or unsorted input; the latter strategy uses an internal hashtable. Notice the lack of any direct information about the aggregate function to be computed. They are found by scanning the node's tlist and quals during executor startup. [4, plannodes.h, 766–778]

```
typedef struct Agg
{
  Plan        plan;
  AggStrategy aggstrategy;  /* PLAIN | SORTED | HASHED | MIXED */
  int         numCols;      /* number of grouping columns */
  AttrNumber *grpColIdx;    /* their indexes in the target list */
  Oid        *grpOperators; /* equality operators to compare with */
  Bitmapset  *aggParams;    /* IDs of Params used in Aggref inputs */
} Agg;
```

Listing 2.10: Aggregate structure definition [4, plannodes.h, 780–793]

2.1.10 Join

The superclass of all join operators is *Join*. This structure holds information about the join type, for example if it is an *INNER* or *OUTER* join.

When the join type is `INNER`, `joinqual` and `plan.qual` are semantically interchangeable. This is not the case for `OUTER` joins. `inner_unique` is set if the `joinquals` are such that no more than one inner tuple could match any given outer tuple. This allows the executor to skip searching for additional matches. [4, plannodes.h, 645–662]

```
typedef struct Join
{
  Plan      plan;
  JoinType  jointype;
  bool      inner_unique;
  List     *joinqual;    /* JOIN quals (in addition to plan.qual) */
} Join;
```

Listing 2.11: Join structure definition [4, plannodes.h, 780–793]

2.1.11 Nested Loop Join

The *Nested Loop* operator is used to perform a join between two tables. A `Nested Loop` operator requires two input sets (given that a `Nested Loop` joins two tables, this makes perfect sense).

`Nested Loop` works by fetching each row from one of the input sets (called the *outer table*). For each row in the outer table, the other input (called the *inner table*) is searched for a row that meets the join qualifier. [2, p. 156]

The nestParams list identifies any executor parameters that must be passed into execution of the inner subplan carrying values from the current row of the outer subplan. Currently PostgreSQL restricts these values to be simple Vars. [4, plannodes.h, 672–681]

```
typedef struct NestLoop
{
  Join  join;
  List *nestParams; /* list of NestLoopParam nodes */
} NestLoop;

typedef struct NestLoopParam
{
  NodeTag  type;
  int      paramno;  /* number of the PARAM_EXEC Param to set */
  Var     *paramval; /* outer-relation Var to assign to Param */
} NestLoopParam;
```

Listing 2.12: Nested Loop Join structure definition [4, plannodes.h, 683–694]

2.1.12 Merge Join

The *Merge Join* operator also joins two tables. Like the `Nested Loop` operator, `Merge Join` requires two input sets: an outer table and an inner table. Each input set must be ordered by the join columns.

`Merge Join` starts reading the first row from each table. If the join columns are equal, `Merge Join` creates a new row containing the necessary columns from each input table and returns the new row. `Merge Join` then moves to the next row in the outer table and joins it with the corresponding row in the inner table. [2, pp. 156–157]

```
typedef struct MergeJoin
{
  Join  join;
  List *mergeclauses;    /* mergeclauses as expression trees */
  int  *mergeStrategies; /* per-clause ordering (ASC or DESC) */
  bool *mergeNullsFirst; /* per-clause nulls ordering */
} MergeJoin;
```

Listing 2.13: Merge join structure definition [4, plannodes.h, 683–694]

2.1.13 Hash and Hash Join

> The *Hash* and *Hash Join* operators work together. The `Hash Join` operator requires two input sets, again called the outer and inner tables. [...]
>
> Unlike other join operators, `Hash Join` does not require either input set to be ordered by the join column. Instead, the inner table is *always* a hash table, and the ordering of the outer table is not important.
>
> The `Hash Join` operator starts by creating its inner table using the `Hash` operator. The `Hash` operator creates a temporary Hash index that covers the join column in the inner table.
>
> Once the hash table (that is, the inner table) has been created, `Hash Join` reads each row in the outer table, hashes the join column (from the outer table), and searches the temporary Hash index for a matching value. [2, p. 158]

```
typedef struct HashJoin
{
  Join  join;
  List *hashclauses;
} HashJoin;

typedef struct Hash
{
  Plan plan;
  Oid  skewTable;        /* outer join key's table OID, or InvalidOID */
  AttrNumber skewColumn; /* outer join key's column #, or zero */
}
```

Listing 2.14: Hash join structure definition [4, plannodes.h, 723–727]

3 PHYSICAL PLAN INJECTION

To use the PostgreSQL executor independently of SQL, an interface that allows loading, execution and storage of execution plans is required. PostgreSQL offers nothing comparable by default. Anyway, an extension can be used to implement such an interface.

3.1 Plan Tree printing and parsing

The PostgreSQL internal module defined in the file `outfuncs.c` can be used to *serialize* a plan tree into a *human readable* textual representation. Every node that can appear in a plan tree must have an output function there. The module defines the function `char* nodeToString(const void* obj)` which takes a plan tree as input and generates the corresponding string representation. Fig. 3.1 shows a truncated example of a serialized plan.

In addition to the output function, every node must have an input function in the file `read.c` as well. The function `void* stringToNode(char* str)` implements the *deserialization* of the string representation and generates the corresponding internal node.

The following applies: `stringToNode(nodeToString(node))` ≡ `node`

However, the `stringToNode` function does not perform any *sanity checks*. This is problematic because the string representation of a faulty plan will be parsed unnoticed as long as the syntax is correct. Normally, only PostgreSQL internal modules use the `stringToNode` function. Therefore, an error-free input is expected and no error messages are provided if the parsing fails.

The PostgreSQL executor relies on the planner/optimizer to create error-free plans. There are several `Assert`-statements implemented for each operator to support the development process, but they are deactivated by default. Therefore, the executor is highly prone to errors in the `plan tree` structure.

It is possible to deserialize an arbitrary execution plan by using `stringToNode`, but it would be extremely hard to develop or modify such a plan without assistance.

```
{PLANNEDSTMT
    :commandType 1
    :parallelModeNeeded false
    :planTree
        {LIMIT
        :parallel_aware false
        :parallel_safe false
        :plan_node_id 0
        :targetlist [...]
        }
    :nParamExec 0
}
```

```
typedef struct PlannedStmt
{
  CmdType    commandType;
  . . .
  bool   parallelModeNeeded;
  struct Plan *planTree;
  int    nParamExec;
} PlannedStmt;
```

Figure 3.1: Left: Textual representation of a plan tree. Right: PlannedStmt data definition

A user would have to know every field of any node in the correct ordering and format. It is very difficult to do this without making mistakes for non-trivial plans.

3.2 Execution of custom plan trees

PostgreSQL is designed to run SQL queries. To our knowledge, there is no side entrance that allows the execution of plan trees directly. Figure 2.1 shows the basic processing steps that a query passes through. The stages before the executor are not required for the execution of a (custom) plan tree. PostgreSQL implements several methods to execute a query, using the Server Programming Interface (SPI) for example, but none of these functions allows to skip the steps prior to the executor.

> The SPI gives writers of user-defined C functions the ability to run SQL commands inside their functions. SPI is a set of interface functions to simplify access to the parser, planner, and executor. [3, Chapter 46. Server Programming Interface]

Instead of writing a custom execution method from scratch, the extensive set of *hooks* provided by PostgreSQL can be used to inject a physical `plan tree` into the executor. Each hook consists of a global function pointer that allows to modify or replace the behavior of a module, without having to rebuild core code. A hook can be enabled and disabled by setting a global variable.

The `planner_hook` allows to replace the `standard_planner` with another function (see Lst. 3.2). A planner that ignores any input and statically returns an arbitrary

```
PlannedStmt *
planner(Query *parse, int cursorOptions, ParamListInfo boundParams)
{
     PlannedStmt *result;

     if (planner_hook)
         result = (*planner_hook) (parse, cursorOptions, boundParams);
     else
         result = standard_planner(parse, cursorOptions, boundParams);
     return result;
}
```

Listing 3.2: Query optimizer entry point [4, planner.c, 255–265]

execution plan will be used to perform the plan injection. Using the `planner_hook` low level error- and memory management can be avoided, because the same execution method as usual can be used. This also makes it possible to use a forced plan as part of a regular query.

The injection can now be done easily. After parsing an execution plan using the `stringToNode` function, for example SPI can be used to issue an arbitrary query (`SELECT 1;`) while the `planner_hook` is enabled. PostgreSQL will parse, analyze and rewrite this query as usual. After the analyze and rewrite module, the *query tree* is passed on to the planner/optimizer. Usually, the standard planner would now generate an execution plan for this query. But because the `planner_hook` is enabled, our own planner is used instead of the standard planner which returns the previously parsed execution plan. This plan gets evaluated by the executor accordingly. The injection is done at this point.

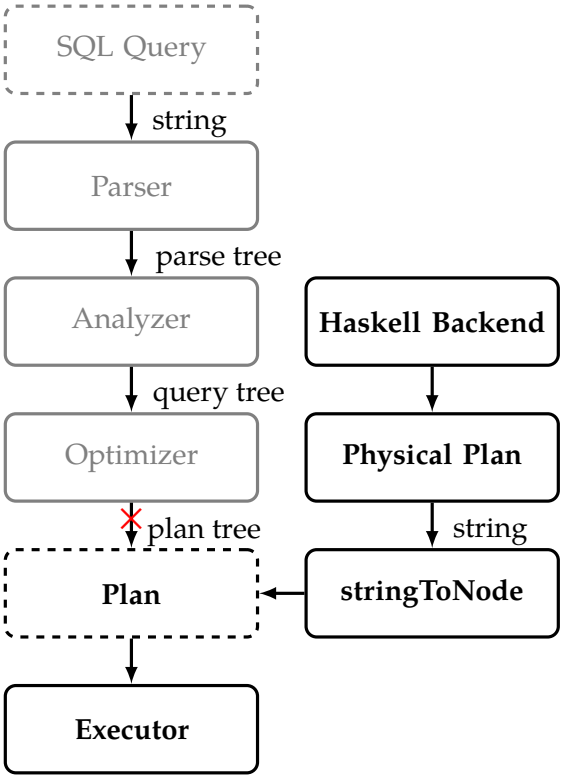

Figure 3.3: Basic steps PostgreSQL performs

4

HASKELL BACKEND

The plan tree data structure contains any information that the executor needs to evaluate a query. This makes these structures verbose and confusing. Fig. 1 of the appendix shows the complete plan tree of the trivial query `SELECT 1` which already spans over 60 lines of code. Normally, the user would not care about all of this information, because the RDBMS automatically determines the required information based on the SQL query. We have to determine all this information manually, because we want to use the executor directly and independently of SQL code. The amount of information and the required accuracy makes it extremely difficult to write an execution plan manually.

Throughout this thesis we will consider the `plan tree` data structures as a Domain Specific Language (DSL). The *syntax* of this language is defined by the serialization format, which seems to be easy initially, but it contains many keywords and some special cases. Therefore, a tool that supports the process of creating artificial physical plans is required.

Haskell is a good choice for the development of DSLs, because it has a lightweight syntax and it is easy to define domain specific data types. Two DSLs will be used to represent `plan trees`. The first DSL will be referred to as *PgPlan*. This is a replication of the `plan tree` nodes that are defined in the file `plannodes.h`. `PgPlan` will be the result of a *compilation/inference* step. The `PgPlan` DSL can be serialized into the textual `plan tree` format.

The second DSL is called *InAST* and defines the language to insert physical plans. In contrast to the `PgPlan` DSL, `InAST` does not contain any data fields, that can be deduced by using the Abstract Syntax Tree (AST) of the plan or PostgreSQL *catalog information*. This removes numerous data fields and thus considerably reduces the amount of information to be entered. That way, the user can focus on the semantics of a plan tree, instead of the syntax. Fig. 2 of the appendix shows the required input to create the `plan tree` as shown in fig. 1 of the appendix. Just 5 fields have to be entered instead of 51 in order to express the same `plan tree`. The remaining values can be deduced automatically.

There is no automated way to identify which data fields can be removed from

the `InAST` DSL. The PostgreSQL extension described in chapter 3 can be used to investigate arbitrary plan trees and develop knowledge about the data fields that way. For each field it has to be determined if the correct value can be deduced using either the AST or catalog information. If this is the case, that field can be removed from the `InAST` DSL and the value will be deduced during the inference phase.

The Haskell back-end uses an *inference rule system* to compile an `InAST` plan into a `PgPlan`. The inference rules use the catalog tables and the tree structure of the `InAST` plan to deduce implicit data fields. The design of this rule system is a key part of this thesis.

4.1 Haskell Modules

`InAST` defines the DSL that serves as user input. These data structures allow the creation of physical plans.

Typically, the type system of Haskell would be used to restrict the data structures, so that only valid plans can be constructed. This is not possible in this case, because the plan structures of PostgreSQL are defined too generically. The plan trees contain some implicit and context-sensitive assumptions, that can not be expressed using simple Haskell data structures. Therefore, the data structures of `InAST` are defined generically too.

`PgPlan` defines the second DSL. These AST structures will be instantiated by the inference rules.

`Validator` implements a simple validation of `InAST` plans at run time. Several sanity checks are performed during a full traversal of the InAST structures. This is useful to detect corner cases which would result in a faulty plan. However, this module does not perform a full *semantical* analysis.

`Extract` traverses the InAST and collects the range table entries as well as all constants. The extraction of this information before the inference starts makes the inference rules easier.

`Inference` implements the rule system that transforms an `InAST` plan into a `PgPlan`. The individual transformation rules will be presented in the form of *inference rules*.

`Pretty Printer` is used to generate the string serialization of `PgPlan`. The generated string will be parsed by PostgreSQL.

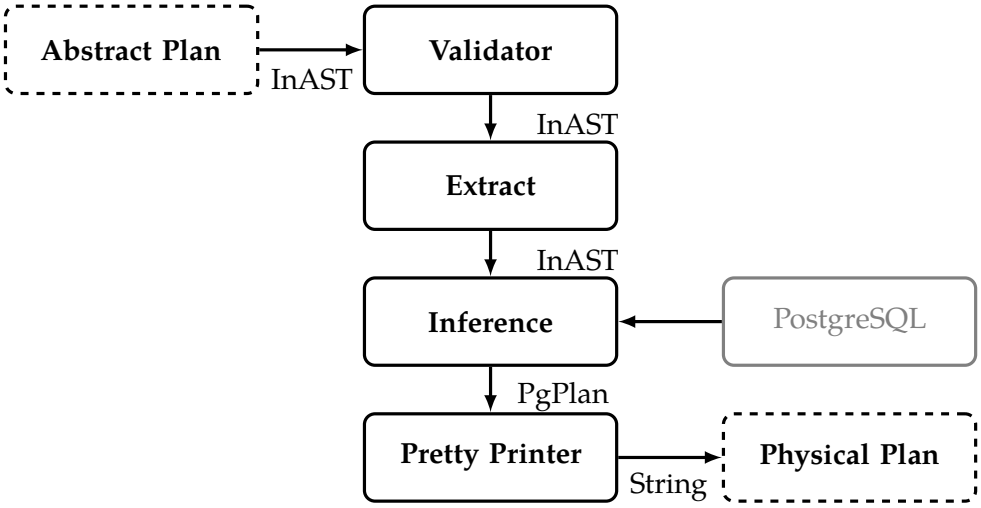

Figure 4.1: A visualization of the sequence in which the Haskell modules are used.

4.2 PostgreSQL System Catalogs

"The system catalogs are the place where a relational database management system stores schema metadata, such as information about tables and columns, and internal bookkeeping information. PostgreSQL's system catalogs are regular tables. You can drop and recreate the tables, add columns, insert and update values, and severely mess up your system that way." [3, 51. System Catalogs]

Table 4.4 shows the system catalogs that the inference rule system uses.

4.2.1 Example usage of System Catalog information

```
A.OPEXPR                        CONST
  { A.oprname="<"               { consttype   = 23
  , A.oprargs=                  , consttypmod = -1
    [ A.CONST "1" "int4"        , constcollid = 0
    , A.CONST "42" "int4"       , constlen    = 4
    ]                           , constbyval  = True
  }                             , constisnull = False
                                , constvalue  = Seq 4 [1, 0, 0, 0, 0, 0, 0, 0]
                                }
```

Figure 4.2: The expression on the left shows the input AST of the operator. The `CONST` on the right is the translation of the first argument.

This section shows exemplary how the inference rule system uses system catalogs. An `OPEXPR` expression will be used as an example here. Other rules use different system catalogs but the basic principle stays the same. The complete inference rule of `OPEXPR` will be shown in section 4.3.2.5.

Suppose that we want to compile the operator expression of fig. 4.2, which equates to the SQL code `1 < 42`. There exist several "<"-operators that are distinguishable by the types of their arguments. The information about existing operators is stored in `pg_operator` and in `pg_proc`.

The operator's arguments must be compiled first. Every expression that is relevant for plan trees would usually have been processed by the `analyze` module of PostgreSQL and therefore must have a *type annotation*. This annotation is a positive integer value which corresponds to a *unique object identifier* of the *pg_type* catalog table. To give an example: the OID of the data type `int4` is 23.

Because the arguments' types are known after compilation, the following SQL query can be used to find the correct operator OID:

```
SELECT o.oid, p.oid AS procoid, . . . , p.proretset, p.prorettype
FROM   pg_operator as o, pg_proc as p
WHERE     o.oprname  = '<'
      AND o.oprleft  = 23
      AND o.oprright = 23
      AND p.oid = o.oprcode;
```

A standard instance of PostgreSQL 10 returns the following result:

oid	procoid	...	proretset	prorettype
97	66	...	f	16

```
typedef struct OpExpr
{
    . . .
    Oid         opno;         /* PG_OPERATOR OID of the operator */
    Oid         opfuncid;     /* PG_PROC OID of underlying function */
    Oid         opresulttype; /* PG_TYPE OID of result value */
    bool        opretset;     /* true if operator returns set */
    . . .
} OpExpr;
```

Listing 4.3: Data structure of OpExpr [4, primnodes.h, 493–504]

Catalog	Description
`pg_attribute`	Stores information about table columns. There is exactly one row for every column of every table in the database
`pg_class`	Catalogs tables and most everything else that has columns or is otherwise similar to a table. This includes indexes.
`pg_index`	Contains part of the information about indexes. The rest is mostly in `pg_class`.
`pg_operator`	Stores information about operators
`pg_proc`	Stores information about functions (or procedures)
`pg_type`	Stores information about data types

Table 4.4: Catalog information used by the inference rules. [3, 51.* System Catalogs]

4.3 Inference rules

This section shows the inference rule system used to deduce the data fields we removed from the `InAST` DSL.

The symbol Γ consists of multiple tables. Each table holds context information during the deduction process.

Γ_{rtable} holds information about tables and their columns.

$\Gamma_{parameters}$ is an integer used to count the number of parameters.

Γ_{pg_*} is used to lookup information in the respective PostgreSQL catalog table.

$\Gamma_{constants}$ constants transformed by using the PostgreSQL extension.

There are two classes of inference rules:

$\mapsto$ is used to transform an `InAST` operator into an `PgPlan` operator

$\rightarrowtail$ is used to transform expressions

4.3.1 Rules for operators

4.3.1.1 PlannedStmt

`PlannedStmt` is the root node of every execution plan. The majority of data fields of `PlannedStmt` are used to properly setup the executor for the query. Several data fields of this node can be ignored, because the rule system is restricted to `select`-statements.

`parallelModeNeeded` must be set to `True` in order to evaluate a parallel execution plan. Otherwise, PostgreSQL will execute the plan sequentially. The user does not have to set this field explicitly, because it can be deduced using the AST.

$$\frac{p_1 \mapsto p_1' \ \ldots \ p_n \mapsto p_n' \qquad \Gamma \vdash o \mapsto o'}{\text{params} \equiv \Gamma(\text{nParamExec}) \quad \text{parallelMode} \equiv \Gamma(\text{parallelModeNeeded})}$$

```
PlannedStmt                    ↦ PLANNEDSTMT
{ plantree = o                   { parallelModeNeeded = parallelMode
, subplans = [p1,...,pn]         , planTree           = o'
}                                , subplans           = [p'1,...,p'n]
                                 , nParamExec         = params
                                 }
```

4.3.1.2 Seq Scan

As stated in section 2.1.3, `Seq Scan` requires a `rtable`-list entry. This list is generated before the inference starts and can be accessed using the context of the computation. We have to find the respective index for the `Seq Scan`.

$$\frac{\Gamma \vdash tl \rightarrowtail tl' \qquad \Gamma \vdash qs \rightarrowtail qs' \qquad \Gamma_{\text{rtable}}(r) \mapsto r_{id}}{\begin{array}{ll} \texttt{SEQSCAN} & \mapsto \texttt{SEQSCAN} \\ \texttt{\{ targetlist = } tl & \quad \texttt{\{ targetlist = } tl' \\ \texttt{, qual = } qs & \quad \texttt{, qual = } qs' \\ \texttt{, scanrelation = } r & \quad \texttt{, scanrelid = } r_{id} \\ \texttt{\}} & \quad \texttt{\}} \end{array}}$$

4.3.1.3 Sort

For each column used as a sort key, a comparison operator for the corresponding data type is required. Ascending sorting requires the operator "<", descending sorting ">" accordingly. `pg_operator` is used to find the appropriate operator.

$$\frac{\begin{array}{l} \Gamma \vdash outer\_plan \mapsto outer\_plan' \\ \Gamma \cup (\text{"OUTER\_VAR"}, outer\_plan') \vdash tl \rightarrowtail tl' \\ oprname(True) \equiv \text{"<"} \qquad\qquad oprname(False) \equiv \text{">"} \\ \Gamma_{\text{pg\_operator}}\big(\texttt{typeof}(tl'[id_1]), oprname(asc_1)\big) = op_1 \\ \ldots \\ \Gamma_{\text{pg\_operator}}\big(\texttt{typeof}(tl'[id_n]), oprname(asc_n)\big) = op_n \end{array}}{\begin{array}{ll} \texttt{SORT} & \mapsto \texttt{SORT} \\ \texttt{\{ targetlist = } tl & \quad \texttt{\{ targetlist = } tl' \\ \texttt{, operator = } outer\_plan & \quad \texttt{, numCols = } n \\ \texttt{, sortCols =} & \quad \texttt{, sortColIdx = } [id_1, \ldots, id_n] \\ \quad \texttt{[} (id_1, asc_1, nulls_1) & \quad \texttt{, sortOperators = } [op_1, \ldots, op_n] \\ \quad \texttt{, } \ldots & \quad \texttt{, nullsFirst = } [nulls_1, \ldots, nulls_n] \\ \quad \texttt{, } (id_n, asc_n, nulls_n) & \quad \texttt{, operator = } outer\_plan' \\ \quad \texttt{]} & \quad \texttt{\}} \\ \texttt{\}} & \end{array}}$$

4.3.1.4 Limit

$$\frac{\Gamma \vdash outer\_plan \mapsto outer\_plan' \qquad \Gamma \vdash x \rightarrowtail x' \qquad \Gamma \vdash y \rightarrowtail y'}{\begin{array}{ll} \texttt{LIMIT} & \mapsto \texttt{LIMIT} \\ \texttt{\{ operator = } outer\_plan & \quad \texttt{\{ operator = } outer\_plan' \\ \texttt{, limitOffset = } x & \quad \texttt{, limitOffset = } x' \\ \texttt{, limitCount = } y & \quad \texttt{, limitCount = } y' \\ \texttt{\}} & \quad \texttt{\}} \end{array}}$$

4.3.1.5 Unique

`Unique` requires similar information as `Sort` does. The main difference is, that we need the equality operator for each data type.

$$\frac{\begin{array}{l}\Gamma \vdash outer\_plan \mapsto outer\_plan' \\ \Gamma_{\texttt{pg\_operator}}(\texttt{typeof}(tl'[id_1]), \texttt{"="}) = op_1 \\ \ldots \\ \Gamma_{\texttt{pg\_operator}}(\texttt{typeof}(tl'[id_n]), \texttt{"="}) = op_n\end{array}}{\begin{array}{ll}\texttt{UNIQUE} & \mapsto \texttt{UNIQUE} \\ \texttt{\{ operator} = outer\_plan & \quad \texttt{\{ numCols} = n \\ \texttt{, uniqueCols} = \texttt{[} id_1, \ldots, id_n \texttt{]} & \quad \texttt{, uniqColIdx} = \texttt{[}id_1, \ldots, id_n\texttt{]} \\ \texttt{\}} & \quad \texttt{, uniqOperators} = \texttt{[}op_1, \ldots, op_n\texttt{]} \\ & \quad \texttt{, operator} = outer\_plan' \\ & \quad \texttt{\}}\end{array}}$$

4.3.1.6 Aggregate

$$\frac{\begin{array}{l}\Gamma \vdash outer\_plan \mapsto outer\_plan' \\ \Gamma \cup (\text{"OUTER\_VAR"}, outer\_plan') \vdash tl \rightarrowtail tl' \\ \Gamma_{\texttt{pg\_operator}}\big(\texttt{typeof}(tl'[id_1]), \texttt{"="}\big) = op_1 \\ \ldots \\ \Gamma_{\texttt{pg\_operator}}\big(\texttt{typeof}(tl'[id_n]), \texttt{"="}\big) = op_n\end{array}}{\begin{array}{ll}\texttt{AGG} & \mapsto \texttt{AGG} \\ \texttt{\{ targetlist} = tl & \quad \texttt{\{ targetlist} = tl' \\ \texttt{, operator} = outer\_plan & \quad \texttt{, numCols} = n \\ \texttt{, groupCols} = \texttt{[}id_1, \ldots, id_n\texttt{]} & \quad \texttt{, grpColIdx} = \texttt{[}id_1, \ldots, id_n\texttt{]} \\ \texttt{\}} & \quad \texttt{, grpOperators} = \texttt{[}op_1, \ldots, op_n\texttt{]} \\ & \quad \texttt{, operator} = outer\_plan' \\ & \quad \texttt{\}}\end{array}}$$

4.3.1.7 Hash

$$\frac{\Gamma \vdash outer\_plan \mapsto outer\_plan'}{\begin{array}{ll}\texttt{HASH} & \mapsto \texttt{HASH} \\ \texttt{\{ lefttree} = outer\_plan \texttt{ \}} & \quad \texttt{\{ lefttree} = outer\_plan' \texttt{ \}}\end{array}}$$

4.3.1.8 Nested Loop Join

A nested loop join can *bind* parameters from the outer-plan, which then can be used by the inner-plan. The value of this parameter is updated for each new row of the outer-plan. This way, correlated queries can be expressed.

$$\frac{\begin{array}{l}\Gamma \vdash outer\_plan \mapsto outer\_plan' \qquad \Gamma \vdash inner\_plan \mapsto inner\_plan' \\ \Gamma^+ \equiv \Gamma \cup (\text{"OUTER\_VAR"}, outer\_plan') \cup (\text{"INNER\_VAR"}, inner\_plan') \\ \Gamma^+ \vdash tl \rightarrowtail tl' \\ \Gamma^+ \vdash q_1 \rightarrowtail q_1' \quad \ldots \quad \Gamma^+ \vdash q_n \rightarrowtail q_n' \\ \Gamma^+ \vdash e_1 \rightarrowtail e_1' \quad \ldots \quad \Gamma^+ \vdash e_m \rightarrowtail e_m' \\ trJoinType(\text{INNER}) \equiv 0 \quad trJoinType(\text{LEFT}) \equiv 1 \quad trJoinType(\text{FULL}) \equiv 2 \\ trJoinType(\text{RIGHT}) \equiv 3 \quad trJoinType(\text{SEMI}) \equiv 4 \quad trJoinType(\text{ANTI}) \equiv 5 \\ joinType' \equiv trJoinType(joinType) \\ currParams = \Gamma^+(nParamExec) \qquad \Gamma^+(nParamExec) = currParams + m\end{array}}{\begin{array}{ll}\texttt{NESTLOOP} & \mapsto \texttt{NESTLOOP} \\ \texttt{\{ targetlist} = tl & \quad \texttt{\{ targetlist} = tl' \\ \texttt{, joinType} = \texttt{joinType} & \quad \texttt{, joinType} = \texttt{joinType'} \\ \texttt{, joinquals} = [q_1, \ldots, q_n] & \quad \texttt{, joinquals} = [q_1', \ldots, q_n'] \\ \texttt{, nestParam} = [e_1, \ldots, e_m] & \quad \texttt{, nestParams} = [e_1', \ldots, e_m'] \\ \texttt{, lefttree} = outer\_plan & \quad \texttt{, lefttree} = outer\_plan' \\ \texttt{, righttree} = inner\_plan & \quad \texttt{, righttree} = inner\_plan' \\ \texttt{\}} & \quad \texttt{\}}\end{array}}$$

4.3.1.9 Hash Join

$$\frac{\begin{array}{l}\Gamma \vdash outer\_plan \mapsto outer\_plan' \qquad \Gamma \vdash inner\_plan \mapsto inner\_plan' \\ \Gamma^+ \equiv \Gamma \cup (\text{"OUTER\_VAR"}, outer\_plan') \cup (\text{"INNER\_VAR"}, inner\_plan') \\ \Gamma^+ \vdash tl \rightarrowtail tl' \\ \Gamma^+ \vdash q_1 \rightarrowtail q_1' \quad \ldots \quad \Gamma^+ \vdash q_n \rightarrowtail q_n' \\ \Gamma^+ \vdash e_1 \rightarrowtail e_1' \quad \ldots \quad \Gamma^+ \vdash e_m \rightarrowtail e_m' \\ trJoinType(\text{INNER}) \equiv 0 \quad trJoinType(\text{LEFT}) \equiv 1 \quad trJoinType(\text{FULL}) \equiv 2 \\ trJoinType(\text{RIGHT}) \equiv 3 \quad trJoinType(\text{SEMI}) \equiv 4 \quad trJoinType(\text{ANTI}) \equiv 5 \\ joinType' \equiv trJoinType(joinType)\end{array}}{\begin{array}{ll}\texttt{HASHJOIN} & \mapsto \texttt{HASHJOIN} \\ \texttt{\{ targetlist} = tl & \quad \texttt{\{ targetlist} = tl' \\ \texttt{, joinType} = \texttt{joinType} & \quad \texttt{, joinType} = \texttt{joinType'} \\ \texttt{, joinquals} = [q_1, \ldots, q_n] & \quad \texttt{, joinquals} = [q_1', \ldots, q_n'] \\ \texttt{, hashclauses} = [e_1, \ldots, e_m] & \quad \texttt{, hashclauses} = [e_1', \ldots, e_m'] \\ \texttt{, lefttree} = outer\_plan & \quad \texttt{, lefttree} = outer\_plan' \\ \texttt{, righttree} = inner\_plan & \quad \texttt{, righttree} = inner\_plan' \\ \texttt{\}} & \quad \texttt{\}}\end{array}}$$

4.3.2 Rules for expressions

4.3.2.1 VAR

Var is an expression node representing a variable, for example a table column. `Var` can also be used to point to the outputs of subplans. For example, in a join node *varTable* becomes `"INNER_VAR"` or `"OUTER_VAR"`. `"INDEX_VAR"` is used to identify Vars that reference an index column rather than a heap column. [4, primnodes.h, 137–152]

$$\frac{\begin{array}{l}\Gamma_{rtable}(t) = t' \\ index(\text{"INNER\_VAR"}) \equiv 65000 \\ index(\text{"INDEX\_VAR"}) \equiv 65002 \\ index(t'(tName)) = tIdx \\ c'(cAtttypid) = varTID \\ c'(cAttcollation) = varColl\end{array} \qquad \begin{array}{r} t'(c) = c' \\ index(\text{"OUTER\_VAR"}) \equiv 65001 \\ index(table) \equiv t'(position) \\ c'(cAttnum) = varIdx \\ c'(cAtttypmod) = varTMod \end{array}}{\begin{array}{ll}\textbf{VAR} & \rightarrowtail \textbf{VAR} \\ \texttt{\{ varTable = } t & \texttt{\{ varno = } tIdx \\ \texttt{, varColumn = } c & \texttt{, varattno = } varIdx \\ \texttt{\}} & \texttt{, vartype = } varTID \\ & \texttt{, vartypmod = } varTMod \\ & \texttt{, varcollid = } varColl \\ & \texttt{\}}\end{array}}$$

4.3.2.2 VARPOS

The PostgreSQL executor uses the position of a column to point to the output of subplans. We need *VARPOS* as additional expression, because it is not obligatory for columns of subplans to have a name. *VARPOS* is essentially the same expression as `VAR` but the index of the column is given instead of the name.

$$\frac{\begin{array}{l}\Gamma_{rtable}(t) = t' \\ index(\text{"INNER\_VAR"}) \equiv 65000 \\ index(\text{"INDEX\_VAR"}) \equiv 65002 \\ c'(cAtttypid) = varTID \\ c'(cAttcollation) = varColl\end{array} \qquad \begin{array}{r} t'(c) = c' \\ index(\text{"OUTER\_VAR"}) \equiv 65001 \\ index(table) \equiv t'(position) \\ c'(cAtttypmod) = varTMod \\ c'(cAttnum) = varIdx \end{array}}{\begin{array}{ll}\textbf{VARPOS} & \rightarrowtail \textbf{VAR} \\ \texttt{\{ varTable = } t & \texttt{\{ varno = } tIdx \\ \texttt{, varPos = } tIdx & \texttt{, varattno = } varIdx \\ \texttt{\}} & \texttt{, vartype = } varTID \\ & \texttt{, vartypmod = } varTMod \\ & \texttt{, varcollid = } varColl \\ & \texttt{\}}\end{array}}$$

4.3.2.3 CONST

The inference of constants is challenging because PostgreSQL uses an abstract representation for constant data types called *Datum*. PostgreSQL supports constants for numerous data types, and we would have to implement a conversion method for each of them. This would be tedious and error prone because we would have to understand the representation of every data type. Also, we would not be able to support user defined data types that way. Instead of doing that, we use the parse and analyze module of PostgreSQL to do the work for us. That way we can ensure that our constants are transformed correctly.

$$\frac{\begin{array}{ll}\Gamma_{constants}(v,t) = c & \\ c(consttype) = tID & c(consttypmod) = tMod \\ c(constcollid) = tColl & c(constlen) = tLen \\ c(constisnull) = tIsNull & c(constvalue) = v'\end{array}}{\begin{array}{ll}\texttt{CONST} & \rightarrowtail \texttt{CONST} \\ \texttt{\{ constvalue = } v & \texttt{\{ consttype = } tId \\ \texttt{, consttype = } t & \texttt{, consttypmod = } tMod \\ \texttt{\}} & \texttt{, constcollid = } tColl \\ & \texttt{, constlen = } tLen \\ & \texttt{, constisnull = } tIsNull \\ & \texttt{, constvalue = } v' \\ & \texttt{\}}\end{array}}$$

4.3.2.4 FUNCEXPR

A FUNCEXPR is used whenever a function is called or a cast is required. The user has to provide the name of the function as well as the parameters. The appropriate function of the table pg_proc is then selected.

$$\frac{\begin{array}{lll}\Gamma \vdash a_1 \rightarrowtail a'_1 & \ldots & \Gamma \vdash a_n \rightarrowtail a'_n \\ \Gamma_{pg\_proc}(f) = row_f & & row_f(OID) = funcid \\ row_f(prorettype) = funcrettype & & row_f(proretset) = funcretset\end{array}}{\begin{array}{ll}\texttt{FUNCEXPR} & \rightarrowtail \texttt{FUNCEXPR} \\ \texttt{\{ funcname = } f & \texttt{\{ funcid = } funcid \\ \texttt{, funcargs = [}a_1,\ldots,a_n\texttt{]} & \texttt{, funcresulttype = } funcrettype \\ \texttt{\}} & \texttt{, funcretset = } funcretset \\ & \texttt{, args = [}a'_1,\ldots,a'_n\texttt{]} \\ & \texttt{\}}\end{array}}$$

4.3.2.5 OPEXPR

`OPEXPR` is used to express operators, similar to `FUNCEXPR`. Operators in PostgreSQL are *syntactic sugar* and could be removed without loosing functionality and expressive power. Every operator in the table `pg_operator` must have a corresponding function in `pg_proc` which performs the actual computation.

$$\frac{\begin{array}{lll}\Gamma \vdash a_1 \rightarrowtail a_1' & \ldots & \Gamma \vdash a_n \rightarrowtail a_n' \\ \Gamma_{pg\_operator}(op) = oprow_{op} & & \Gamma_{pg\_proc}(oprow_{op}(oprcode)) = row_{op} \\ oprow_{op}(oid) = opid & & row_{op}(oid) = funcid \\ row_f(prorettype) = funcrettype & & row_f(proretset) = funcretset\end{array}}{\begin{array}{ll}\texttt{OPEXPR} & \rightarrowtail \texttt{OPEXPR} \\ \texttt{\{ oprname = } op & \texttt{\{ opno = } opid \\ \texttt{, oprargs = [}a_1,\ldots,a_n\texttt{]} & \texttt{, opfuncid = } funcid \\ \texttt{\}} & \texttt{, opresulttype = } funcrettype \\ & \texttt{, funcretset = } funcretset \\ & \texttt{, args = [}a_1',\ldots,a_n'\texttt{]} \\ & \texttt{\}}\end{array}}$$

4.3.2.6 PARAM

A parameter can be used for one of two things in a plan tree. First, a value can be supplied from outside the plan. Such parameters are numbered from 1 to n. Second, executor internal parameters can be used for passing values into and out of a sub-query or from nestloop joins to their inner scans. For historical reasons, such parameters are numbered from 0. These numbers are independent of external parameters. [4, primnodes.h, 207–232]

$$\frac{\begin{array}{lll}\Gamma_{pg\_type}(t) = t' & & t'(oid) = tID \\ kind(\text{"EXTERNAL"}) \equiv 0 & kind(\text{"EXEC"}) \equiv 1 & k' = kind(k)\end{array}}{\begin{array}{ll}\texttt{PARAM} & \rightarrowtail \texttt{PARAM} \\ \texttt{\{ paramid = } id & \texttt{\{ paramid = } id \\ \texttt{, paramtype = } t & \texttt{, paramtype = } tID \\ \texttt{, paramkind = } k & \texttt{, paramkind = } k' \\ \texttt{\}} & \texttt{\}}\end{array}}$$

5 EXPERIMENTS

5.1 Unnesting of correlated subqueries

According to Thomas Neumann and Alfons Kemper, unnesting of *correlated subqueries* can greatly speed up query processing. They found, that no existing RDBMS can de-correlate queries in the general case. However, they provide a generic approach for unnesting arbitrary queries based on an algebraic representation of a query with correlated subqueries. The unnesting is performed using a rule system. [6]

In chapter 1, Neumann and Kemper present the two nested SQL queries Q_1 and Q_2, see lst. 5.1 and lst. 5.3. In the subsequent chapters, they have been able to show that their rule system is able to produce an unnested version of these queries. The corresponding logical plans of Q_1 and Q_2 can be seen in lst. 5.1 and lst. 5.3.

To demonstrate the importance of their approach, they compared the unnesting capabilities of various RDBMS, including PostgreSQL 9.1 and *HyPer*. HyPer is a RDBMS that has been developed by Neumann and Kemper originally. The query planner/optimizer of HyPer uses their generic unnesting approach and was therefore able to unnest Q_2 whereas PostgreSQL 9.1 was not. The impact of their unnesting approach on PostgreSQL is not clear yet.

Neumann and Kemper did measure the execution time of Q_1 and Q_2 using PostgreSQL 9.1, which was unable to unnest these query. We observed the same behavior with PostgreSQL 10. However, instead of using a SQL query as input, we can now *force* the corresponding logical plans to determine the impact of unnesting for PostgreSQL.

Just as Neumann and Kemper have, we used $1,000$ `student` and $10,000$ `exam` tuples as test data. We then reproduced their experiment for Q_1 and Q_2 using PostgreSQL 10. We will not reproduce their TPC-H experiment because "every vendor makes sure that the well known TPC-H queries are correctly unnested in their system". [6]

We created physical execution plans based on the logical plans for Q_1 and Q_2. Following this, we measured the execution time for the unnested plans as well as for the plans that PostgreSQL generates for the corresponding SQL-query.

```
select s.name, e.course
from   students s, exams e
where  s.id=e.sid and
       e.grade=(select min(e2.grade)
                from exams e2
                where s.id=e2.sid);
```

Listing 5.1: Nested query Q_1 from [6]

```
select s.name,e.course
from   students s,exams e,
       (select e2.sid as id, min(e2.grade) as best
        from exams e2
        group by e2.sid) m
where  s.id=e.sid and m.id=s.id and e.grade=m.best;
```

Listing 5.2: Decorrelated version of Q_1 from [6]

The nested version of Q_1 had a run time of 3926.69 ms whereas the decorrelated formulation could be executed in 28.97 ms. The physically mapped logical plan as shown in fig. 5.6 took 47.77 ms to execute. The decorrelated version of Q_1 did perform slightly better than the Neumann and Kemper plan, but the significance of unnesting is indisputable. However, the *physical operator mapping* is ambiguous in general, thus there might be an even better physical plan.

Neumann and Kemper were unable to test their logical plan for Q_2 on any system besides HyPer, because no other system was able to unnest this query. This thesis makes this test now possible. PostgreSQL 10 needs $17,199.45$ ms without unnesting and $3,745.55$ ms with unnesting to execute the plans.

We created the SQL query of lst. 5.4 based on the decorrelated logical plan of Q_2. The execution times of this query and the forced physical plan are identical because PostgreSQL generated the same execution plan.

The performance improvements match those observed by Neumann and Kemper and therefore further supports the idea "that a system should be able to unnest arbitrary queries". [6]

Fig. 5.8 shows the most efficient physical plan for the logical plan of Q_2.

```
SELECT s.name, e.course
FROM students s, exams e
WHERE s.id=e.sid and
    (s.major = 'CS' or s.major = 'Games Eng') and
    e.grade>=(SELECT avg(e2.grade)+1 --one grade worse
              FROM exams e2 --than the average grade
              WHERE s.id=e2.sid or --of exams taken by
              (e2.curriculum=s.major and --him/her or taken
              s.year>e2.date)); --by elder peers
```

Listing 5.3: Q_2 from [6]

```
SELECT a.name, a.course
FROM ( SELECT *
       FROM students s, exams e
       WHERE (s.major='CS' OR s.major='Games Eng')
             AND s.id=e.sid ) a,
     ( SELECT d.id, d.year, d.major, avg(e2.grade) as m
       FROM ( SELECT *
              FROM students s, exams e
              WHERE (s.major='CS' OR s.major='Games Eng')
                    AND s.id=e.sid ) d
            , exams e2
       WHERE d.id=e2.sid OR
             (d.year > e2.date AND (e2.curriculum=d.major))
       GROUP BY d.id, d.year, d.major ) as d
WHERE a.grade >= m+1 AND
      (d.id = a.id OR ( d.year > a.date AND (a.curriculum=d.major) ));
```

Listing 5.4: Decorrelated version of Q_2

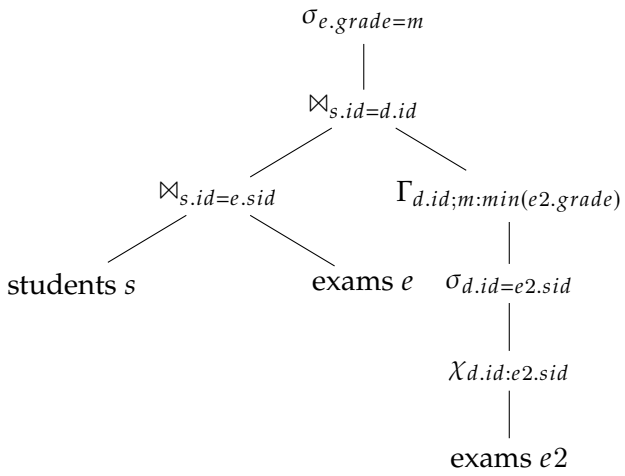

Figure 5.5: Logical plan of Q_1

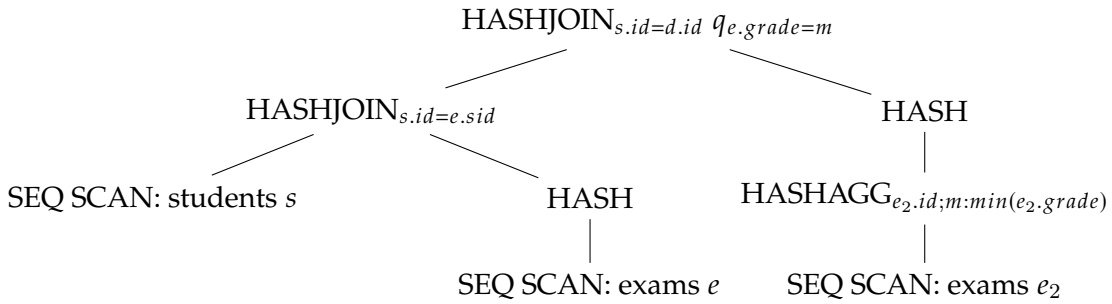

Figure 5.6: Physical plan of Q_1 for PostgreSQL

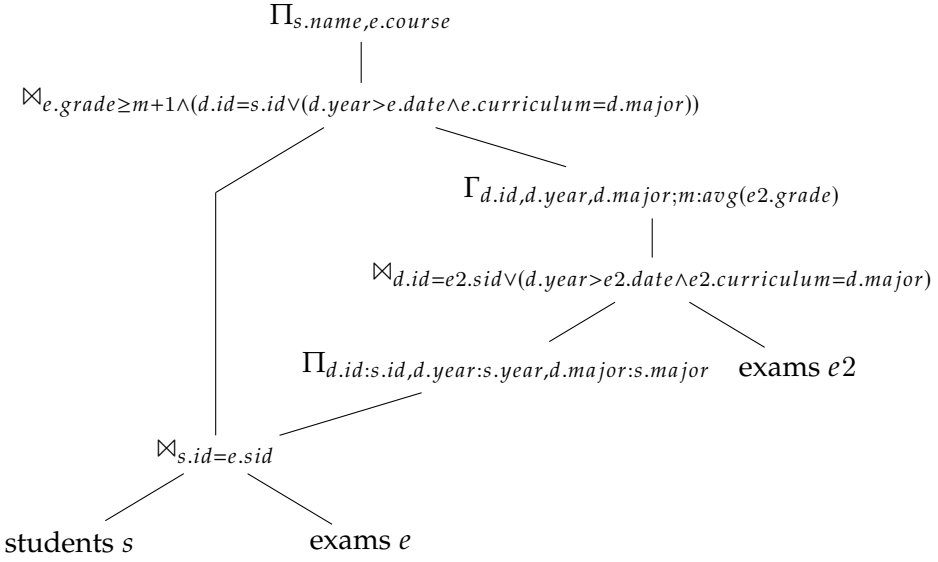

Figure 5.7: Logical plan of Q_2

```
Nested Loop
  Output: students.name, exams.course
  Join Filter: ((exams.grade >= int4(avg(exams.grade)))
                AND ((students.id = students.id)
                     OR ((students.year > exams.date)
                     AND (exams.curriculum = students.major))))
  ->  Hash Join
        Hash Cond: (students.id = exams.sid)
        ->  Seq Scan on public.students
              Filter: ((students.major = 'CS'::text)
                       OR (students.major = 'Games Eng'::text))
        ->  Hash
              ->  Seq Scan on public.exams
  ->  Materialize
        ->  HashAggregate
              Output: students.id, students.year, students.major, avg(exams.grade)
              Group Key: students.id, students.year, students.major
              ->  Hash Join
                    Hash Cond: (students.id = exams.sid)
                    Join Filter: ((students.id = exams.sid)
                                  OR ((students.year > exams.date)
                                  AND (students.major = exams.curriculum)))
                    ->  Hash Join
                          Hash Cond: (students.id = exams.sid)
                          ->  Seq Scan on public.students
                                Filter: ((students.major = 'CS'::text)
                                         OR (students.major = 'Games Eng'::text))
                          ->  Hash
                                ->  Seq Scan on public.exams
                    ->  Hash
                          ->  Seq Scan on public.exams
```

Figure 5.8: Physical plan of Q_2 using `EXPLAIN`

6 CONCLUSION

The Haskell back-end that we presented as inference rules in chapter 4 makes the construction of artificial physical plans significantly easier, but extensive knowledge about the PostgreSQL executor is still required.

The `PgPlan` DSL combined with the usage of the pretty printing module ensures the *syntactical* correctness of artificial execution plans. However, the DSL does not prevent the formulation of *semantically* wrong plans. Despite the fact that the `Validator` module performs several sanity checks, faulty plans can not be detected reliable. It is hard to debug semantically wrong plans, because the PostgreSQL executor tends to crash without reporting useful error messages. One way to resolve such a problem is to use a debugger such as *LLDB*, but a PostgreSQL server with debug symbols is required. The debugger can be used to find the source code fragment where PostgreSQL messed up. Usually, it is then possible to trace back which parameter of the plan induced the error, but it is a tedious task because the serialization of a plan becomes very big for non-trivial plans.

6.1 Related Work

Some RDBMS such as Oracle, MariaDB and SQL Server natively support *optimizer hints* that can be used with SQL statements to alter execution plans. "Hints let you make decisions usually made by the optimizer. Hints provide a mechanism to instruct the optimizer to choose a certain query execution plan based on the specific criteria. For example, you might know that a certain index is more selective for certain queries. Based on this information, you might be able to choose a more efficient plan than the optimizer." [8, 16 Using Optimizer Hints]

Plan hints are a powerful but also *dangerous* tool. They are relatively easy to use because they utilize annotated SQL code, but they restrict the optimizer's ability to make choices. Wrong usage can result in a catastrophic run-time.

> Appropriate use of the right hint on the right query can improve query performance. The exact same hint used on another query can create

> more problems than it solves, radically slowing your query and leading to serve blocking and timeouts in your application. [7, p. 177]
>
> While query hints allow you to control the behavior of the optimizer, it doesn't mean your choices are necessarily better than the optimizer's choices. [...] Also, remember that a hint applied today may work well but, over time, as data and statistics shift, the hint may no longer work as expected. [7, p. 181]

Plan hints do not allow as much manipulation of an execution plan as plan forcing does, but they are not as error prone. Generating invalid execution plans happens quickly when each parameter can and has to be set explicitly.

SQL Server 2005 introduced the `USE PLAN` query hint. This allows to specify an entire execution plan as a target to be used to optimize a query. The `USE PLAN` hint can force most of the specified plan properties, including the tree structure, join order, join algorithms, aggregations, sorting, unions, and index operations like scans. [7, pp. 248–250] This is very similar to the approach presented in chapter 3 of this thesis. In contrast to this thesis, SQL Server does not provide a tool that would make it easier to construct execution plans directly.

6.2 Future Work

While most of the relational operators are implemented, the majority of SQL expressions are not. PostgreSQL 10 supports the major features of SQL:2011. [3, Appendix D. SQL Conformance] The outcome of this is a very rich set of expressions. It is not difficult to formulate inference rules for more expressions, but it is time-consuming.

We think that it is possible to use the result of this thesis to implement *the generic approach for unnesting arbitrary queries* by Thomas Neumann and Alfons Kemper for PostgreSQL. [6] Our experiments in chapter 5 prove the significance of query unnesting.

APPENDIX

```
{PLANNEDSTMT
:commandType 1
:queryId 0
:hasReturning false
:hasModifyingCTE false
:canSetTag true
:transientPlan false
:dependsOnRole false
:parallelModeNeeded false
:planTree
   {RESULT
   :startup_cost 0.00
   :total_cost 0.01
   :plan_rows 1
   :plan_width 4
   :parallel_aware false
   :parallel_safe true
   :plan_node_id 0
   :targetlist (
      {TARGETENTRY
      :expr
         {CONST
         :consttype 23
         :consttypmod -1
         :constcollid 0
         :constlen 4
         :constbyval true
         :constisnull false
         :location 7
         :constvalue 4 [ 1 0 0 0 0 0 0 0 ]
         }
      :resno 1
      :resname ?column?
      :ressortgroupref 0
      :resorigtbl 0
      :resorigcol 0
      :resjunk false
      }
   )
   :qual <>
   :lefttree <>
   :righttree <>
   :initPlan <>
   :extParam (b)
   :allParam (b)
   :resconstantqual <>
   }
:rtable <>
:resultRelations <>
:nonleafResultRelations <>
:rootResultRelations <>
:subplans <>
:rewindPlanIDs (b)
:rowMarks <>
:relationOids <>
:invalItems <>
:nParamExec 0
:utilityStmt <>
:stmt_location 0
:stmt_len 0
}
```

Figure 1: Plan tree for query `SELECT 1`

```
RESULT
{ targetlist =
  [ TargetEntry
    { targetexpr = A.CONST "1" "int4"
    , targetresname = "?column?"
    , resjunk = False
    }
  ]
, resconstantqual = Nothing
}
```

Figure 2: User input required to create the plan tree of fig. 1.

Bibliography

[1] Surajit Chaudhuri. "An Overview of Query Optimization in Relational Systems". In: *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. PODS '98. Seattle, Washington, USA: ACM, 1998, pp. 34–43. ISBN: 0-89791-996-3. DOI: 10.1145/275487.275492. URL: http://doi.acm.org/10.1145/275487.275492.

[2] Korry Douglas and Susan Douglas. *PostgreSQL - A Comprehensive Guide to Building, Programming, and Administering PostgresSQL Databases*. Indianapolis, Indiana: Sams Publishing, 2003. ISBN: 978-0-735-71257-7.

[3] The PostgreSQL Global Development Group. *PostgreSQL 10 Documentation*. 2018. URL: http://www.postgresql.org/docs/10/static/index.html (visited on 06/25/2018).

[4] The PostgreSQL Global Development Group. *PostgreSQL 10 Source Code*. 2018. URL: https://www.postgresql.org/ftp/source/v10.0/ (visited on 06/25/2018).

[5] Viktor Leis et al. "How Good Are Query Optimizers, Really?" In: *Proc. VLDB Endow.* 9.3 (Nov. 2015), pp. 204–215. ISSN: 2150-8097. DOI: 10.14778/2850583.2850594. URL: http://dx.doi.org/10.14778/2850583.2850594.

[6] Thomas Neumann and Alfons Kemper. "Unnesting Arbitrary Queries". In: *BTW*. Vol. 241. LNI. GI, 2015, pp. 383–402.

[7] Benjamin Nevarez. *Inside the SQL Server Query Optimizer* -. Red Gate Books, 2011. ISBN: 978-1-906-43460-1.

[8] Oracle. *Oracle Database Online Documentation*. 2018. URL: https://docs.oracle.com/cd/B19306_01/server.102/b14211/hintsref.htm#i8327 (visited on 06/25/2018).

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich diese schriftliche Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe.

Ort, Datum Unterschrift